General instructions

- You have 2 hours to complete the examination. When applicable, people with special facilities have 2h20 minutes in total.
- The exam is "closed book", meaning that you can only make use of the material given to you.
- You are supposed to write the codes in the Python programming language, but syntactic errors are allowed as far as the written algorithm can be well understood.
- **Do not use while statements.**
- **Do not use global variables.**
- The grade is computed with the formula: points/20 $\times$ 9 + 1.
- Keep the names of the variables and functions as stated in the question.
- **If you do not follow these instructions you will not receive any points in the respective question.**
- **Please answer each questions on separate paper sheets.**

Questions

1. Consider the operation $\alpha = x^\top Ax$, for $A \in \mathbb{R}^{n\times n}$ and $x \in \mathbb{R}^n$ (the result is a scalar, $\alpha \in \mathbb{R}$).

   (a) (2 points) Determine the number of operations and the time complexity in Big-O notation (in the form $O(an^p)$, including the constant factor a of the leading term n^p) of computing $x^\top Ax$. Writing a program is not required.

   Solution:

   - (1 point): $y = Ax$: for each row i: n mult, $n-1$ add $\Rightarrow n(2n-1)$
   - (0.5 points): $x^\top y$: n mult, $n-1$ add $\Rightarrow 2n-1$
   - (0.5 points): Total: $2n^2 + n - 1 = O(2n^2)$

   (b) (5 points) Assume that A is a symmetric tridiagonal matrix, i.e., sparse with nonzero entries only on the main diagonal and the first lower and upper diagonals:

$$A = \begin{pmatrix} a_1 & b_1 & & & & \\ b_1 & a_2 & b_2 & & & \\ & \ddots & \ddots & \ddots & & \\ & & b_{n-2} & a_{n-1} & b_{n-1} & \\ & & & b_{n-1} & a_n & \end{pmatrix}.$$

   A is stored in arrays of the diagonal entries $\mathbf{a} := (a_1, \ldots, a_n)$ and the off-diagonal entries $\mathbf{b} := (b_1, \ldots, b_{n-1})$.
   Write a python function `tridiag_xtAx(a, b, x)`, receiving the `numpy` arrays **a** and **b** of the diagonals, and the array **x** of vector x, that computes the vector-matrix-vector product $x^\top Ax$ and returns the resulting number. A must be expressed solely in terms of the arrays $\mathbf{a}, \mathbf{b}$ – **the dense (full) matrix must not be formed!** All `numpy` and `scipy` functions are allowed.

   Solution:

```
# -1 point for wrong function arguments/return (since this is given)
# 0 points total when creating dense numpy matrices from a and b and
# computing x @ A @ x
```

```python
def tridiag_xtAx(a, b, x):
    n = len(x)
    y = np.zeros(n)

    # 0.5 points for splitting the problem in two parts: y = Ax and dot(x, y)

    # compute y = A x
    y[0] = a[0] * x[0] + b[0] * x[1]        # 0.5 points
    y[-1] = b[-1] * x[-2] + a[-1] * x[-1]   # 0.5 points
    # instead they might choose to extend (concatenate) b by 0 and do
    # everything with the loop.
    # this is less efficient, but fine
    for i in range(1, n - 1):               # 0.5 for loop w/ correct limits
                                            # 0 for while loops
        y[i] = b[i-1] * x[i-1] + a[i] * x[i] + b[i] * x[i]   # 1.5 points

    # compute x^T x
    s = 0
    for i in range(n):      # 0.5 points
        s += x[i] * y[i]    # 1 point

    # or simply
    # s = np.dot(x, y)    # (1.5 points)
    return s
```

If a student learned how to use `numpy` and `scipy` efficiently, this should be rewarded:

```python
from scipy import sparse

def tridiag_xtAx(a, b, x):
    A = sparse.diags((b, a, b), (-1, 0, 1)   # 2.5 points
    return x @ A @ x                         # 2.5 points
```

(c) (2 points) Determine the number of operations and the time complexity in Big-O notation (in the form $O(an^p)$, including the constant factor a of the leading term n^p) for the sparse algorithm for computing $x^\top Ax$ devised in (b) for the symmetric tridiagonal matrix A.

Solution:

- (1 point): Ax: $n\times$ 3 multiplications + 2 additions for each row i, minus 1 addition, 1 multiplication for first and last rows, respecitvely $\Rightarrow g(n) = 5n - 4$
- (0.5 points) $x^\top y$: n multiplications, $n - 1$ additions $\Rightarrow h(n) = 2n - 1$
- (0.5 points): Total: $f(n) = 5n - 4 + 2n - 1 = O(7n)$

(d) ($\frac{1}{2}$ point) Assume a dimension of $n = 10^4$. Comparing the time complexity of $x^\top Ax$ with the sparse matrix obtained in (c) with that of the dense matrix in (a), by what factor is the sparse algorithm faster or slower than the dense algorithm, assuming equal fixed cost of all elementary operations?

Solution:

- Dense algorithm: $2 \cdot 10^8$ vs. $7 \cdot 10^4$ sparse operations.
- The sparse algorithm is $\sim \frac{2}{7}10^4$ (≈ 2857) times faster.

(e) (½ point) How many floating point numbers must be stored for a dense $n \times n$ matrix, how many for the sparse symmetric tridiagonal matrix in (b)?

Solution: Free points!
Dense matrix: n^2 floats, sparse matrix: $2n - 1$ floats

Total for Question 1: 0

TURN THE PAGE FOR THE SECOND QUESTION

2. Consider a list of length N containing random integer numbers in the interval [0, 100]. (The following questions can be solved independently.)

   (a) (1½ points) Write a python program that creates the list for a variable number N that is given by the "user" with the `input` function. To create a random integer in the interval $[a, b]$, use the function `randint(a, b)` from the `random` module.

   Solution:

```python
import random   # omitting import is accetable

N = int(input("Enter length of list N: "))   # 0.5 points

numbers = []
for i in range(N):  # 0.25 for correct loop
    numbers.append(random.randint(0, 100))  # 0.5 for list, 0.25 for randint
```

   (b) (6½ points) Write a python function that sorts the list in ascending order and returns the sorted list. The only allowed functions to use are `len` and `range`.

   Solution: One of the simplest solutions: find the smallest entry and put it into position 0 (swap entries). Repeat for all remaining entries.

```python
# 0.5 points for correctly using a function (argument + return)
# the rest must depend on the algorithm they devise.
# 3 points for a working idea (subtract half/full points for logic errors)
# 3 points for correct implementation (subtract half/full points for
# implementation errors that fail the logic to work out)
def sort(a):
    for j in range(len(a) - 1):
        min = a[j]
        i_min = j
        for i in range(j, len(a)):
            if a[i] < min:
                min = a[i]
                i_min = i
        a[i_min] = a[j]
        a[j] = min

    return a
```

   Sorting the list in-place as above or creating a sorted copy is both fine.

   (c) (2 points) Write a python function that receives a *sorted* list and returns a copy of that list with all duplicate entries removed. Do not use existing python functions (such as `set` or `numpy.unique`).

   Solution:

```python
def remove_duplicates(sorted):   # 0.5 for function (args, return)
    filtered = []
    for x in sorted:               # 0.5 for looping through list
        if x not in filtered:      # 0.5 for detecting duplicate
            filtered.append(x)     # 0.5 for handling list
    return filtered
```

   Or, without `in`:

```python
def remove_duplicates(sorted):   # 0.5 for function (args, return)
    filtered = [sorted[0]]
    for i in range(1, len(sorted)):       # 0.5 for looping through list
        if sorted[i] != sorted[i-1]:      # 0.5 for detecting duplicate
            filtered.append(sorted[i])    # 0.5 for handling list
    return filtered
```

Total for Question 2: 0